All Circuits are Busy Now: The 1990 AT&T Long Distance Network Collapse

by

Dennis Burke

CSC440-01
November, 1995
California Polytechnic State University

At 2:25pm on Monday, January 15th, network managers at AT&T's Network Operations Center in Bedminster, N.J. began noticing an alarming number of red warning signals from various parts of their world-wide network. Within seconds, the giant 72 screen video array that graphically represented the network was crisscrossed with a tangle of red lines as a rapidly spreading malfunction leapfrogged from one computer-operated switching center to another. The standard procedures the managers tried first failed to bring the network back up to speed and for nine hours, while engineers raced to stabilize the network, almost 50% of the calls placed through AT&T failed to go through. Until 11:30pm, when network laods were low enough to allow the system to stabilize, AT&T alone lost more than $60 million in unconnected calls. Still unknown is the amount of business lost by airline reservations systems, hotels, rental car agencies and other businesses that relied on the telephone network. This wasn't supposed to happen. AT&T had built a reputation and a huge advertising campaign base on it's reliability and security. What had gone wrong?

On a Good Day

Normally, AT&T's long-distance network was a model of reliability and strength. On any given day, AT&T's long-distance service, which at the time carried over 70% of the nation's long-distance traffic, route over 115 million telephone calls. The backbone of this massive network was a system of 114 computer-operated electronic switches (4ESS) scattered across the United States. These switches, each capable of handling up to 700,000 calls an hour, were linked via a cascading network know[n] as Common Channel Signaling System No. 7. When a telephone call was received by the network from a local exchange, a switch would scan a list of 14 different possible routes to complete the call. At the same time, it passed the telephone number to a parallel signalling network that checked the alternate routes to determine if the switch at the other end could deliver the call to it's local company. If the destination switch was busy, the original switch sent the caller a busy signal and released the line. If the switch was available, a signal-network computer made a reservation at the destination switch and ordered the destination switch to pass the call, after the switches checked to see if the connection was good. The entire process took only four to six seconds.

What Went Wrong

It wook only slightly longer to slow the entire network to a crawl. Working backwards through the data, a team of 100 frantically searching telephone technicians identified the problem, which began in New York City. The New York switch had performed a routine self-test that indicated it was nearing its load limits. As standard procedure, the switch performed a four-second maintenance reset and sent a message over the signalling network that it would take no more calls until further notice. After reset, the New York switch began to distribute the signals that had backed up during the time it was off-line. Across the country, another switch received a message that a call from New York was on it's way, and began to update it's records to show the New York switch back on line. A second message from the New York switch then arrived, lass than ten milliseconds after the first. Because the first message had not yet been handled, the second message should have been saved until later. A software defect then caused the second message to be written over crucial communications information. Software in the receiving switch detected the overwrite and immediately activated a backup link while it reset itself, but another pair of closely timed messages triggered the same response in the backup processor, causing it to shut down also. When the second switch recovered, it began to route it's backlogged calls, and propagated the cycle of close-timed messages and shut-downs throughout the network. The problem repeated iteratively throughout the 114 switches in the network, blocking over 50 million calls in the nine hours it took to stabilize the system.

The Root Problem

The cause of the problem had come months before. In early December, technicians had upgraded the software to speed processing of certain types of messages. Although the upgraded code had been rigorously tested, a one-line bug was inadvertantly added to the recovery software of each of the 114 switches in the network. The defect was a `C` program that featured a `break` statement located within an `if` clause, that was nested within a `switch` clause.

In pseudocode, the program read as follows:

```
1  while (ring receive buffer not empty
          and side buffer not empty) DO

2    Initialize pointer to first message in side buffer
     or ring receive buffer

3    get copy of buffer

4    switch (message)

5       case (incoming_message):

6             if (sending switch is out of service) DO

7                 if (ring write buffer is empty) DO

8                     send "in service" to status map

9                 else

10                    break

                  END IF

11           process incoming message, set up pointers to
             optional parameters

12           break
       END SWITCH


13   do optional parameter work
```

When the destination switch received the second of the two closely timed messages while it was still busy with the first (buffer not empty, line 7), the program should have dropped out of the `if` clause (line 7), processed the incoming message, and set up the pointers to the database (line 11). Instead, because of the break statement in the `else` clause (line 10), the program dropped out of the case statement entirely and began doing optional parameter work which overwrote the data (line 13). Error correction software detected the overwrite and shut the switch down while it couls reset. Because every switch contained the same software, the resets cascaded down the network, incapacitating the system.

Lesson Learned

Unfortunately, it is not difficult for a simple software error to remain undetected, to later bring down even the most reliable systems. The software update loaded in the 4ESSs had already passed through layers of testing adn had remained unnoticed through the busy Christmas season. AT&T was fanatical about it's reliability. The entire network was designed such that no single switch could bring down the system. The software contained self-healing features that isolated defective switches. The network used a system of "paranoid democracy," where switches and other modules constantly monitored each other to determine if they were "sane" or "crazy." Sadly, the Jan. 1990 incident showed the possibility for all of the modules to go "crazy" at once, how bugs in self-healing software can bring down healthy systems, and the difficulty of detecting obscure load- and time-dependent defects in software.

There is still much to be learned from this incident, however. Clearly, the use of `C` programs and compilers contributed to the breakdown. A more structured programming language with stricter compilers would have made this particular defect much more obvious. The routine practice of allowing the long-distance switches to shutdown and reset themselves also contributed. A more fault-tolerant hardware and software system that could handle minor problems without shutting down could have greatly reduced the effects of the defect. The final lesson is a positive one; it is worth noting that with AT&T's careful attention to hardware survivability and extensive testing, this is one of the few problems ever to impact their long-distance network so severely. While the `break` statement flaw could have been avoided with more thorough software engineering techniques, many more problems have already been deterred by the system in place. The AT&T long-distance system crash stands out not just as a software engineering example, but because of it's rarity.

References

Coy, P. & Lewyn, M. (1990, January 29). The day that every phone seemed off the hook. Business Week, pp.39-40.

Gonzalez, D. & Rogers, M. (1990, January 29). Can we trust out software? Newsweek, pp. 70-73.

Joch, A. (1995, December). How software doesn't work. Byte, pp.49-60.

McCarrel, T. & Witterman, P. (1990, January 29). Ghost in the machine. Time, pp. 58-59.

Neumann, P. G. (1995) Computer related risks. New York: ACM Press.

O'Malley, C. (1990, September). When the computer fails. Popular Science, pp. 84-90.

Peterson, I. (1991, February 16). Finding fault. Science News, pp. 104-106.

Wiener, L. R. (1993). Digital woes, why we should not depend on software. New York: Addison-Wesley.